CS221: Digit Recognizer

Min Cheol Kim (mincheol), Katie Hahm (khahm)
December 12, 2014

1. Introduction

Interpreting handwritten digits has been an active area of research in image analysis. Currently, it is widely used in post offices to read addresses and for online banking. Our project will focus on a small aspect of that endeavor. We created an AI that recognizes a single handwritten digit from 0 through 9 and outputs the correct number. Our interest in this subject stems from a competition posted on kaggle [1].

There have been various approaches to this problem, including support vector machines(SVMs), neural networks, convolutional neural networks, and others. One group was able to obtain an error rate of 0.23% on a testing set of 10,000 images using mostly neural networks and convolutional neural networks[2]. For our final project, we wanted to learn about some of these approaches and actually implement them from scratch.

Our dataset is the MNIST dataset provided by the competition. This dataset contains several ten thousands of handwritten digits labeled with the correct number to use as a training dataset. As Y. Le Cun and his colleagues mentioned in his paper, we believe that the inputs of our algorithm can be the raw pixels instead of otherwise pre-processed features[3]. The inputs of our algorithm are a group of upright, grey-scaled image of a single hand-written digit from 0 through 9. We will measure our performance by the percent accuracy in predictions within our testing set.

2. Task Definition/Overview

Because we are attempting to mimic a person classifying the digits, our oracle is a human labeling the images and obtaining their percent accuracy. Our baseline is having the AI randomly choose a digit as an output, which has percent accuracy of approximately 0.1.

Our general procedure is as follows: We take the input as a tuple (pixels, correct label), and the learner will use the training data to output a function. Then, the function is tested with testing data, and we calculate the percent accuracy. Figure 1 illustrates this process.

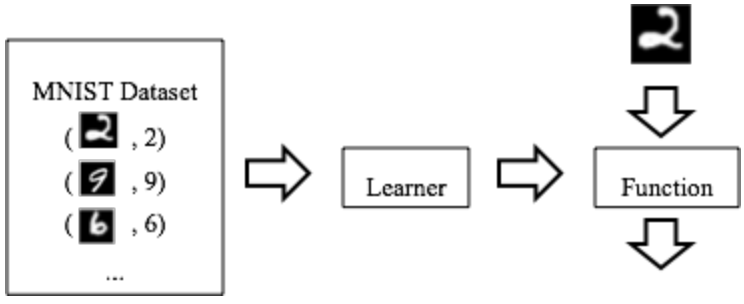

Figure 1: General Procedure

The goal of this project was to design, implement, compare, and contrast various approaches to making this learner. Our first and simplest approach is using a multiclass linear classifier to predict the output data. Then we used a multilayered perceptron, which took more time to run yet yielded more accurate results. Finally, we used an autoencoder to reorganize the pixels and fed that into the linear classifier, which took the most runtime and yielded the worst results.

3. Approaches

3.1 Multiclass Linear Classifier

We designed and implemented a multiclass linear classifier as our simplest approach to this problem. Our model had 10 sets of weights, each corresponding to one of ten output nodes. These output nodes represented digits, and the node that had the best score was taken to be the AI's digit. The input nodes represented the raw pixel values. We used the multiclass hinge loss function given by:

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max_{y'} \{\mathbf{w}_{y'} \cdot \phi(x) - \mathbf{w}_y \cdot \phi(x) + \mathbf{1}[y' \neq y]\}$$

where x corresponds to the input, y the correct label, and $\boldsymbol{w}$ is the set of the 10 weight vectors.The 1[y' = y] term encourages the correct label have higher score by at least one than any of the wrong labels. The gradient of this loss function, as shown in lecture, is equivalent to adding $\varphi(x)$ to $\boldsymbol{w}_{incorrect}$ and subtracting $\varphi(x)$ from $\boldsymbol{w}_{correct}$. For example, if our linear classifier predicted a 2 when the correct label was a 3, we should add $\varphi(x)$ to $\boldsymbol{w}_2$ and subtract $\varphi(x)$ from $\boldsymbol{w}_3$ (to actually update the weights, since we subtract the gradient, we would actually end up adding $\varphi(x)$ to $\boldsymbol{w}_3$ and subtracting $\varphi(x)$ from $\boldsymbol{w}_2$).

After all the weights were trained, the classification decision was made using the following predictor function:

$$\text{Predictor } f_w(x) = \arg\max\{w_y \cdot \varphi(x)\}, y \in \{0, 1, \ldots, 9\}$$

This predictor function simply calculates the score $\mathbf{w} \cdot \varphi(x)$ for all the weights and the classification is the number associated with the weight vector that produces the highest score. Our implementation is demonstrated in figure 2.

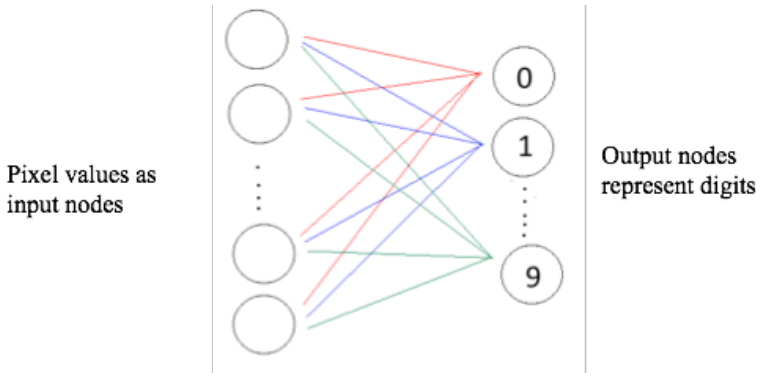

Figure 2. Construction of multiclass linear classifier

We tuned the number of iterations and eta by iteration, and we obtained the highest accuracy at 26 number of iterations and 0.1 eta and achieved a surprisingly good 89.15% accuracy. This method was also the fastest by far of all our approaches. The training history for our best-case multiclass linear classifier is shown in Figure 3.

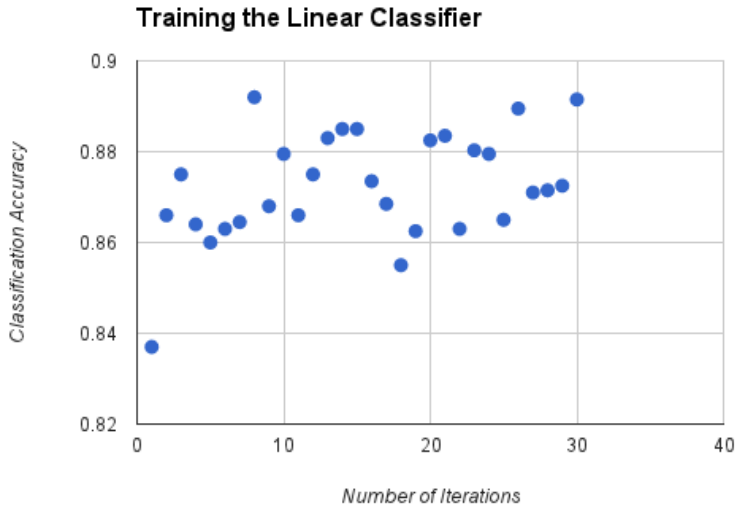

Figure 3. Training the best case linear classifier

3.2 Multilayered Perceptron

Our next approach involved a specific form of a neural network, the multilayered perceptron, a feedforward artificial neural network. The pixel values (normalized to 0.0 - 1.0) themselves served as the input for the neural network (so we had 28 x 28 = 784 input nodes) and we designed the multilayered perceptron to have 10 output nodes, each corresponding to a digit. The inputs and the outputs look similar to those of the multiclass linear classifier, but for the neural network, there were one or more hidden layers in between the input nodes and the output nodes. Our construction of the multilayered perceptron is shown in Figure 4.

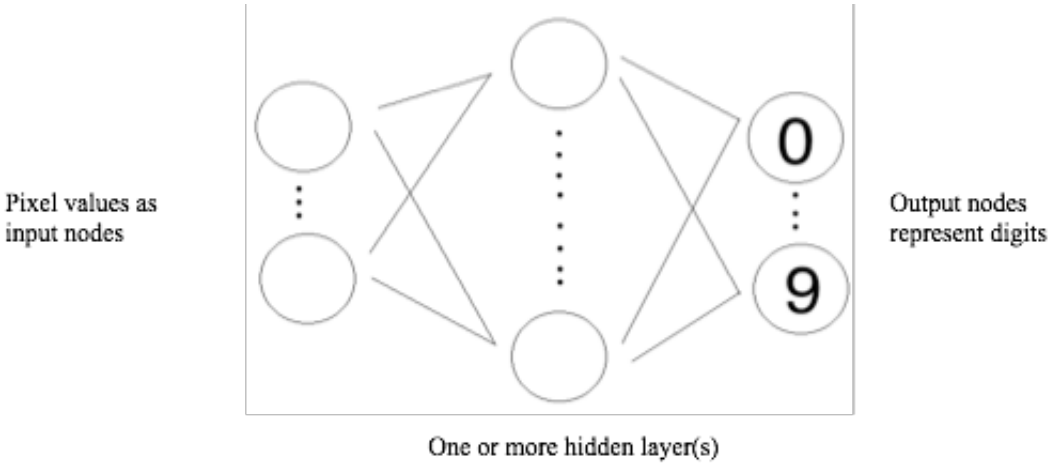

Figure 4. Construction of multilayered perceptron

To feedforward through the network and output values, we used the sigmoid activation function, shown below in Figure 5.

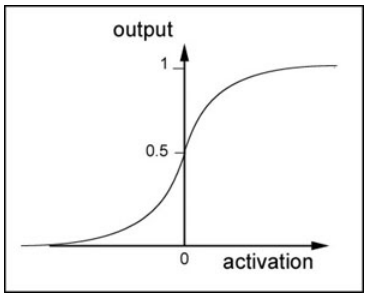

Figure 5. Sigmoid activation function

The weight matrices and the bias vectors for the neural networks were stored in n-dimensional numpy arrays, since numpy had fast implementations of various functions we needed. To feedforward our inputs in the neural network, we iteratively update activations by repeatedly multiplying by the weight matrix, adding the bias, and applying the activation function element-wise.

We decided to use the quadratic cost function, because the derivative was easy to compute compared to some of other available cost functions. The quadratic cost function we used is:

$$Cost = \frac{1}{2*(\#\ training\ examples)} \sum_x ||y - output(x)||^2$$

where y is the correct label in a vector form (for example, if the correct label is a 3, column vector y would be $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^T$ filled with all 0's except the 3rd index).

To backpropagate the errors at the outputs through the neural network, we implemented these steps and equations:

1. Feedforward through the neural network, saving all the z vectors (dot product between weights and the previous activation plus the biases) and activations for each layer
2. Calculate the output error using the equation $\delta_L = \triangle C \odot \sigma'(output\ z\ vector)$, where σ' refers to the derivative of the sigmoid activation function and L refers to the output layer and the $\odot$ symbol refers to the Hadamard product.
3. For all previous layers except the input layer, the errors for each layer can be calculated as $\delta_{layer} = [(weights^{layer+1})^T * \delta_{layer+1}] \odot \sigma'(output\ z\ vector^{layer})$.

Using these calculated errors for each layer, we then implemented mini-batch stochastic gradient descent, where the weights are updated once for one batch of multiple examples.

In addition to having number of iterations and eta as variables, we also had to tune the number hidden layers, number of hidden nodes in each layer, and the mini-batch size. By experimenting, we obtained that the highest accuracy was achieved using 100 hidden nodes, 20 number of iterations, 3.0 eta, and mini-batch size 10, which yielded an accuracy of 96%. Although this

approach ran slower than the linear classifier, the result was much more accurate. Figure 6 shows an example training run of the multilayered perceptron.

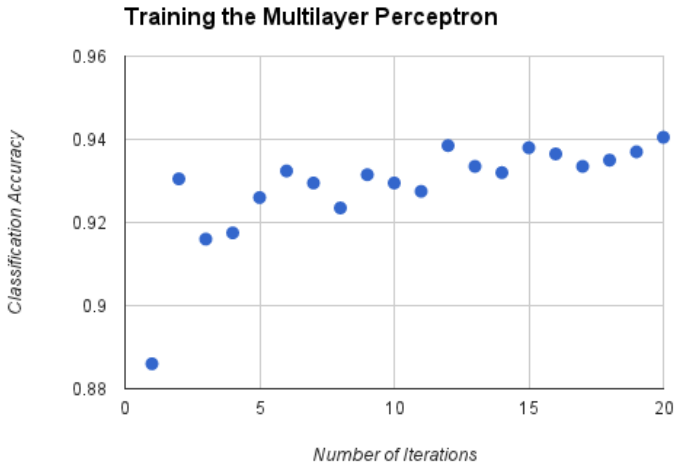

Figure 6. Training the multilayered perceptron

After our initial multilayered perceptron was thoroughly tested, we sought to improve the results and we implemented the L2 regularization to prevent overfitting. This was done by adding a term to the cost function:

$$Cost = \frac{1}{2*(\#\ training\ examples)} \sum_x ||y - output(x)||^2 + \frac{\lambda}{2*(\#\ training\ examples)} \sum_w w^2$$

This means that every time we update the weight matrix, we subtract the gradient not from the weight matrix itself, but a scaled weight matrix by $(1 - \frac{step\ size * \lambda}{\#\ training\ examples})$. With regularization, our best accuracy was achieved by using a single layer of 100 hidden nodes, 40 number of iterations, 3.0 eta, mini-batch size of 10, and lamba of 0.6, which yielded an accuracy of 97.1%. There was a significant increase in our accuracy by preventing overfitting with regularization. The training history of our best-case regularized multilayered perceptron is shown in Figure 7.

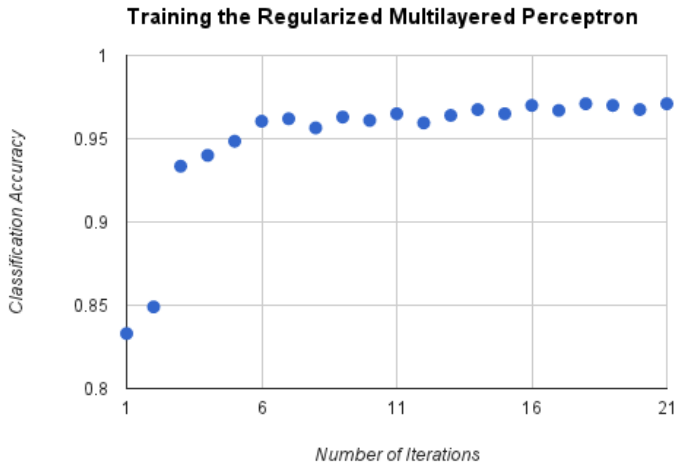

Figure 7. Training the Regularized Multilayered Perceptron

3.3 Autoencoder

Although using the raw pixel values yielded fairly high accuracies, we thought that perhaps we could use a way to pick out features from the raw pixels and use the linear classifier on the chosen features and see if that approach would perform better than running the linear classifier on the raw pixels. We learned that we could use an autoencoder, also a feedforward neural network, to do such automated feature extraction. The basic autoencoder that we implemented this for project has a very similar architecture as a multilayered perceptron.

The key point of the autoencoder is that it has the same number of input nodes and output nodes. The number of hidden nodes is less than the number of input nodes so that the autoencoder is forced to somehow reduce the information in the raw pixels into fewer number of nodes, then use the fewer number of of nodes to re-create our input. This would effectively mean that the hidden nodes represent some features that the autoencoder found important in re-creating the inputs. We can then feed them as the inputs to the previously described linear classifier model. A diagram illustrating this process is shown in figure 8.

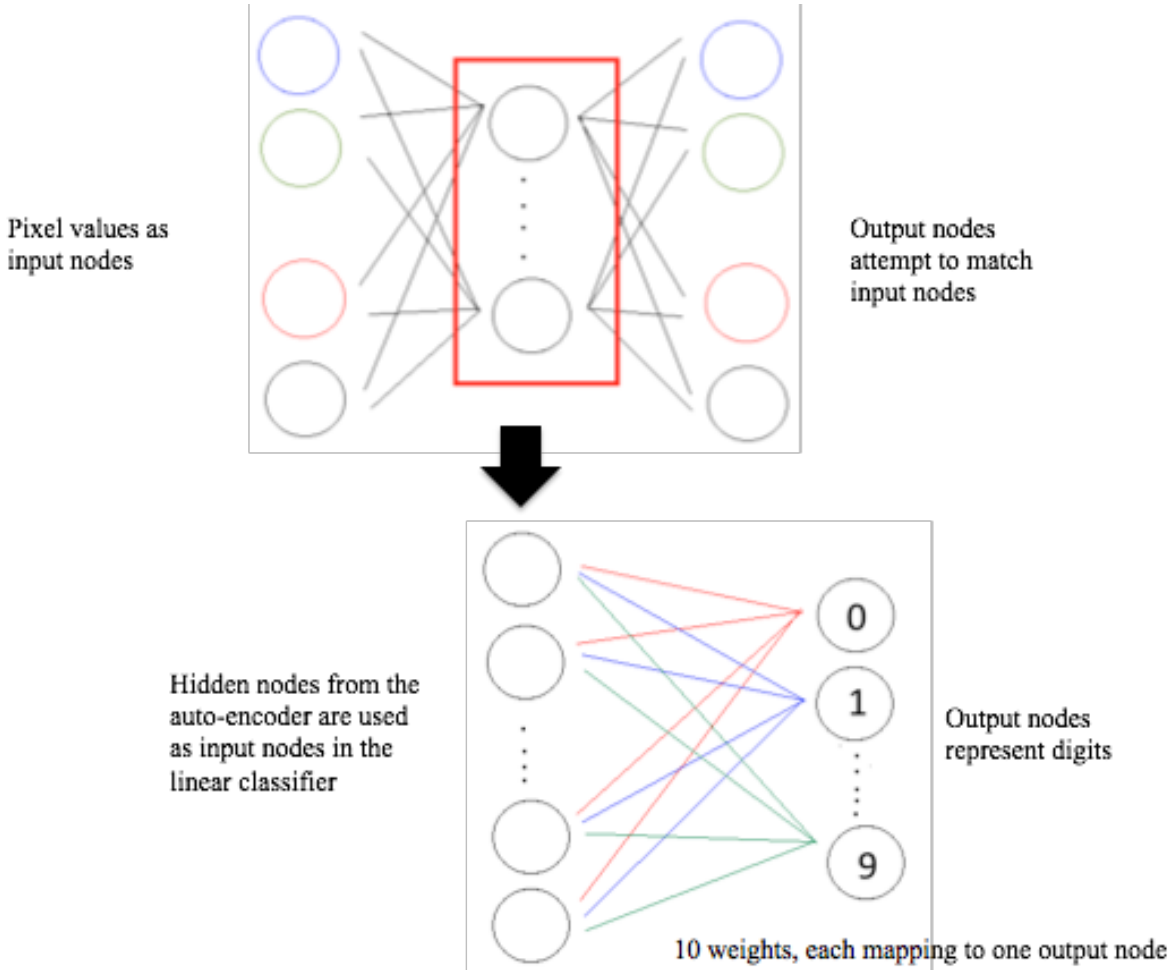

Figure 8. Construction of autoencoder/linear classifier

The autoencoder had the same variables to tune as the multilayered perceptron. The best accuracy was obtained with 400 hidden nodes, 10 number of iterations, 10 batch size, and 3.0 eta. The best performance was surprisingly low - it yielded 87% despite being the slowest to run. The training history of this autoencoder (along with its classification accuracy when paired with a multiclass linear classifier) is shown in Figure 9.

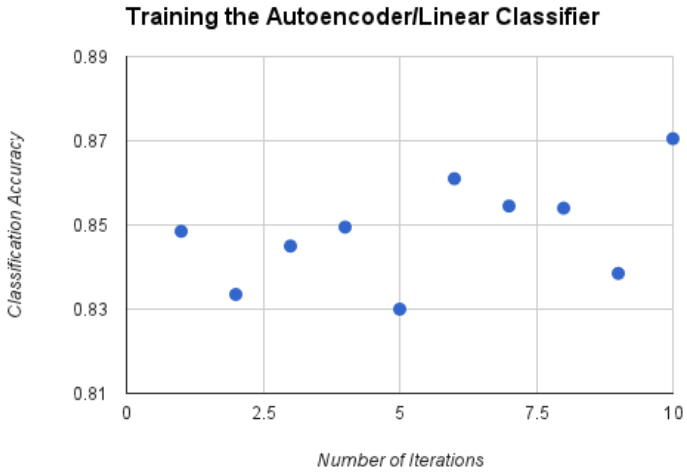

Figure 9. Training the autoencoder

4. Analysis

4.1 Performance Analysis

We were not given the full MNIST dataset; therefore, it was difficult to compare our results to the literature values available for classifying this dataset. We found that using the multiclass linear classifier gave a surprisingly high accuracy despite its simplicity, and that using multilayered perceptron gave a reasonable accuracy (from literature).

However, we found it odd that the autoencoder yielded lower accuracies than the multiclass linear classifier. We believe that this is caused by our context of using supervised and unsupervised learning. Because we had a large set of labeled data, a linear classifier trained on the whole set performed better than a linear classifier being trained on learned features.

4.2 Follow-up Experiment

We conducted a follow-up experiment because we wanted to test whether the autoencoder would perform better in a different environment. We reduced our labeled training data drastically to size 50, while treating the rest of our original data as unlabeled. We obtained this result:

	Linear Classifier	**Autoencoder**
Accuracy	0.54	0.56

The difference is small, but this demonstrates that the use of the autoencoder for feature learning is beneficial under certain environments. We believe that by using more advanced unsupervised learning techniques, such as stacked autoencoders and tied weights in autoencoders, the linear classifier trained on the learned features from an autoencoder would perform better than a linear classifier trained on raw data.

5. Extensions

5.1 Motivation from Post Offices

In practical applications of digit recognizing machines, it is important to achieve a near perfect recognition of handwriting. Time is also an issue in practical applications, as it would be ideal for the computer to read the images quickly. Therefore, we designed and implemented an extension where the linear classifier would reject all images that are uncertain to achieve accuracy and speed. This application was inspired by the processes of post offices.

We approached this problem for the linear classifier by implementing a heuristic. The heuristic was designed to give the confidence level for a certain classification - whether the linear classifier thinks its classification will match the correct label or not. We experimented with various heuristics to filter using the scores. However, our most effective method was:

$$\textit{largest score} > 0.6056 * \textit{sum of positive scores}$$

By doing so, we achieved 100% accuracy, but required the machine to reject most of the dataset. Due to time constraints, we could not explore different designs, but this heuristic yielded our best result.

We obtained this optimal constraint value by iterating over multiple constraint values, decreasing the constraint until the linear classifier obtained 100% accuracy but minimizing the number of rejected images. The false positives (AI was confident yet was wrong) and false negatives (AI was too cautious) plots are shown in figure 10A, 10B.

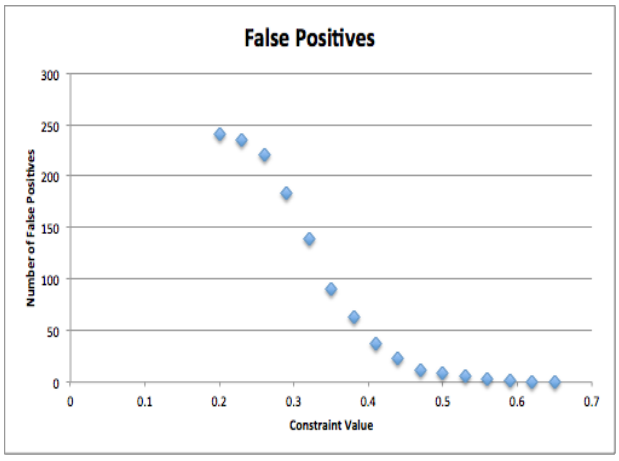

Figure 10A. False Positives vs Constraint

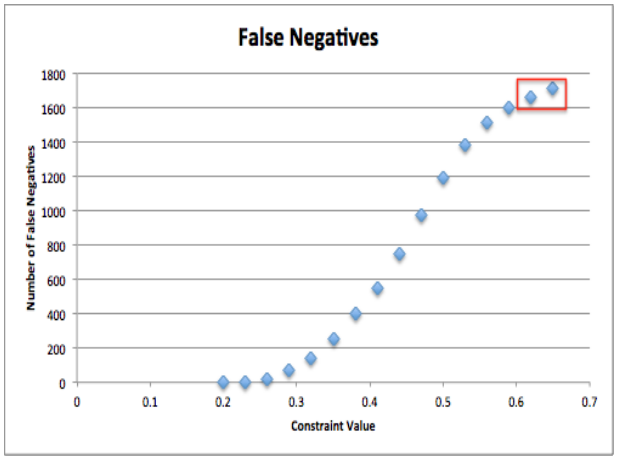

Figure 10B. False Negatives vs Constraint

5.2 Automated Hyperparameter Tuning

Another extension that we implemented was to use automated hyperparameter tuning to find ideal values for the linear classifier. We set the starting number of iterations to 26 and starting eta to 0.5, and the algorithm would try incrementing and decrementing both variables to find higher accuracy results. When it found a higher accuracy result, it would reset its variables to the current and keep incrementing and decrementing from this new state. Although it seemed to converge on the value we found iteratively (26 number of iterations and 0.1 eta), it later jumped to have a slightly higher accuracy of 89.2% with 44 number of iterations and 1.0 eta. This algorithm took much longer to run and we could not obtain large amounts of data.

6. Challenges

We documented some challenges we encountered. Our most prominent challenge was cooperating with speed of execution. Since the MNIST dataset is large, we had to pay close attention to the decisions we made regarding the types of variables, number of loops, and others that may cause time delay. We were especially careful to check that all information we wanted were printed before running iterative code. One solution to speed up the program was to use numpy matrix operations to quicken calculations.

7. References

[1] "Digit Recognizer." *Kaggle*. N.p., 25 July 2012. Web. 12 Dec. 2014.

[2] Ciresan, Dan, Ueli Meier, and Jurgen Schmidhuber. "Multi-column Deep Neural Networks for Image Classification." (n.d.): n. pag. Feb. 2012. Web. 12 Dec. 2014.

[3] Le Cun, Yann, and Et. Al. "Handwritten Digit Recognition with a Back-Propagation Network." (n.d.): n. pag. Web. 12 Dec. 2014.

[4] Various tutorial websites on neural networks, autoencoders, etc. where we learned more about these topics.